Advanced Topics from Scientific Computing
TU Berlin Winter 2024/25
Notebook 05
(cc) BY-SA Jürgen Fuhrmann

Plotting & visualization in Julia

Human perception is much better adapted to visual representation than to numbers

Purposes of plotting:

- Visualization of research results for publications & presentations
- Debugging + developing algorithms
- "In-situ visualization" of evolving computations
- Investigation of data
- 1D, 2D, 3D, 4D data

Julia has several plotting packages. In the moment, IMHO there is not the one package fitting all plotting needs. There are several plotting packages to choose from. Some of them will be covered in this lecture.

Makie

Makie is the evolving native Julia plotting library. Due to the design and complexity of the code, precompilation times are rather long, howeve this much improved with Julia 1.9. It comes in different incarnations which are useful in different situations.

- GLMakie.jl
  - GPU based plotting using modern OpenGL
  - Very good plot performance
  - The Makie to be used when working in the REPL, but also works well in notebooks
- CairoMakie.jl maps Makie API to the Cairo library, which allows for high quality 2D plots in vector (svg) and bitmap (png) formats.
  - works well in notebooks
  - uses the browser for rendering when working from the REPL
  - Limited to 2D
- WGLMakie.jl experimental backend using three.js for work in the browser
- RPRMakie.jl Experimental ray tracing backend using AMDs RadeonProRender.

The different Makie variants have a relatively long precompilation time, which signficantly improved with Julia 1.9.

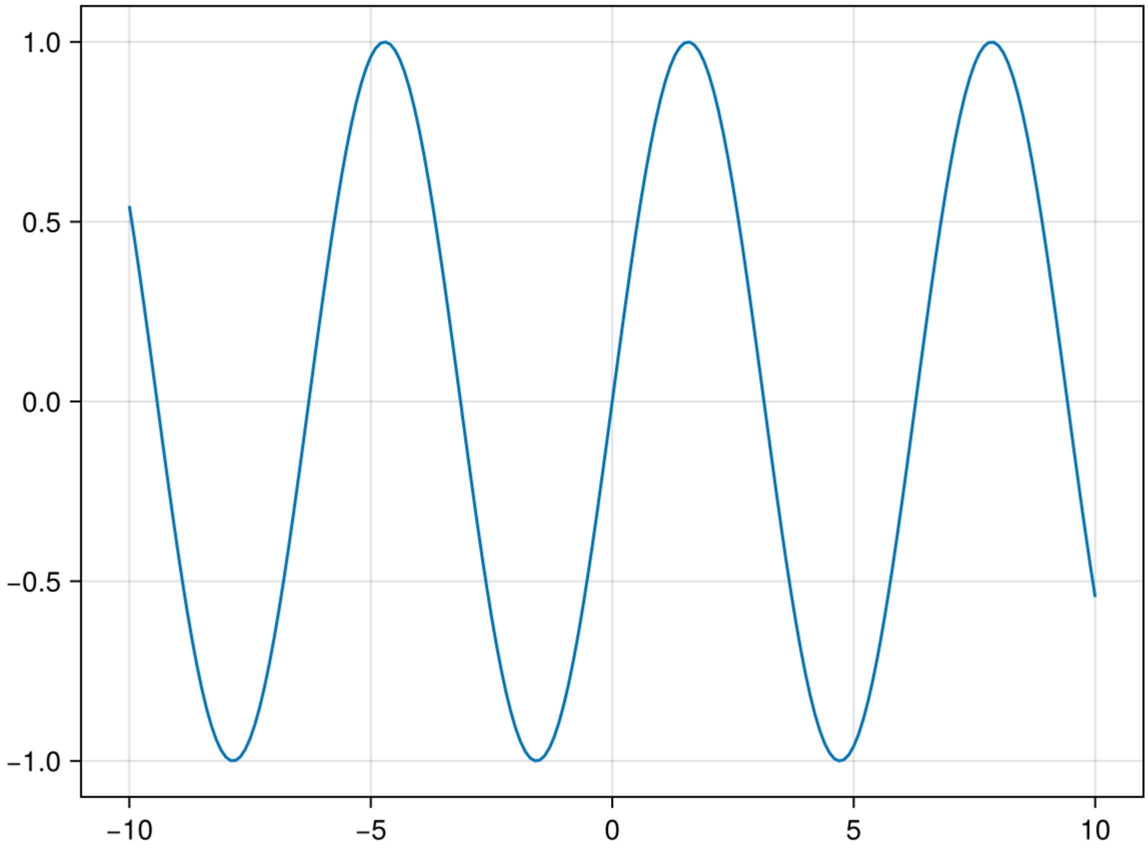
1D plots

Let us create first plots with CairoMakie (see also this blog post):

```
begin
    using CairoMakie
    CairoMakie.activate!(type="png")
end
```

```
X = -10.0:0.1:10.0
```

```
X=-10:0.1:10
```

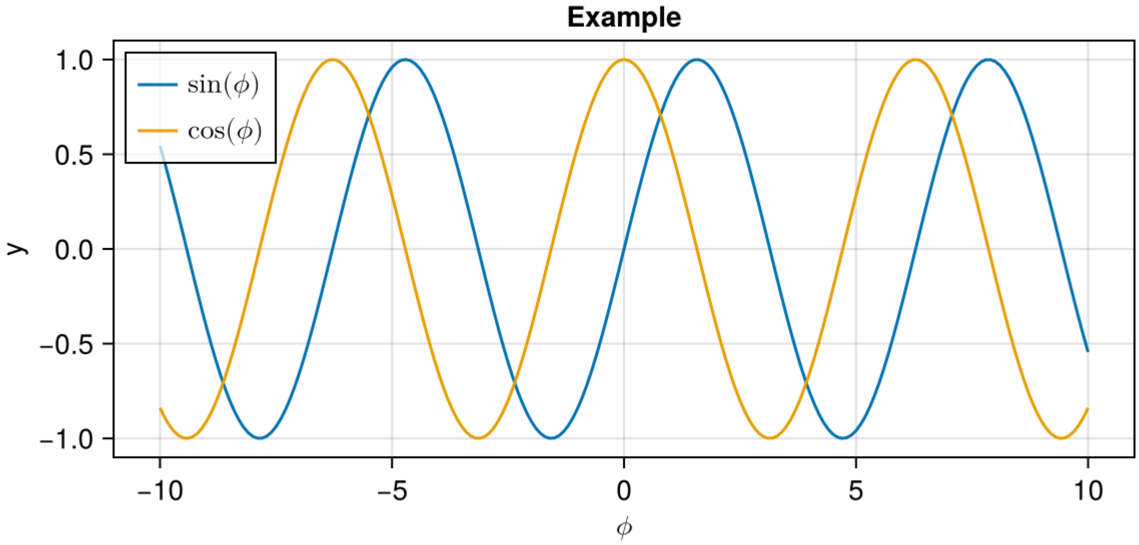


```
lines(X,sin.(X))
```

This plot is not nice. It lacks:

- Title
- Axis labels
- Label of the plot
- Size adjustment

```
using Colors,LaTeXStrings
```

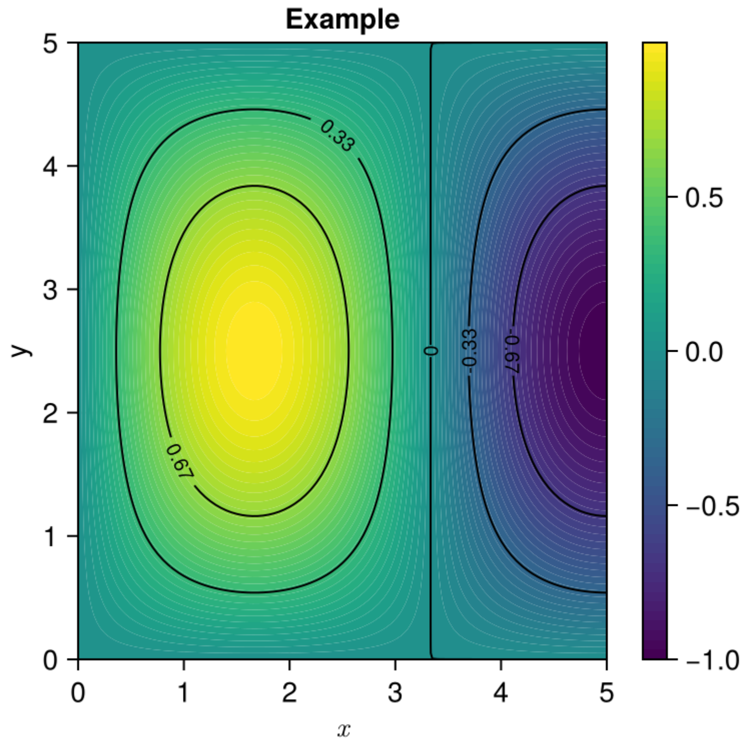


```julia
let
    Φ=-10:0.1:10
    fig=Figure(size=(600,300))
    axis=Axis(fig[1,1],xlabel=L"\phi",ylabel="y",title="Example")
    lines!(axis,Φ,sin.(Φ),label=L"\sin(\phi)")
    lines!(axis,Φ,cos.(Φ),label=L"\cos(\phi)")
    axislegend(axis,backgroundcolor = RGBA(1.0, 1.0, 1.0, 0.7),position=:lt)
    CairoMakie.save("test.pdf",fig)
    fig
end
```

- Instead of a `begin/end` block we used a `let` block. In a let block, all new variables are local and don't interfer with other pluto cells.
- The `backgroundcolor` keyword argument makes the axis legend transparent.
- `position` puts it on a left top position.
- Thanks to the LaTeXStrings package, we can use $\LaTeX$ math strings in plot labels here, we just need to prefix the strings with "L".
- The `save` statement saves the figure to a file. Possible are `svg`, `png`, `pdf`.

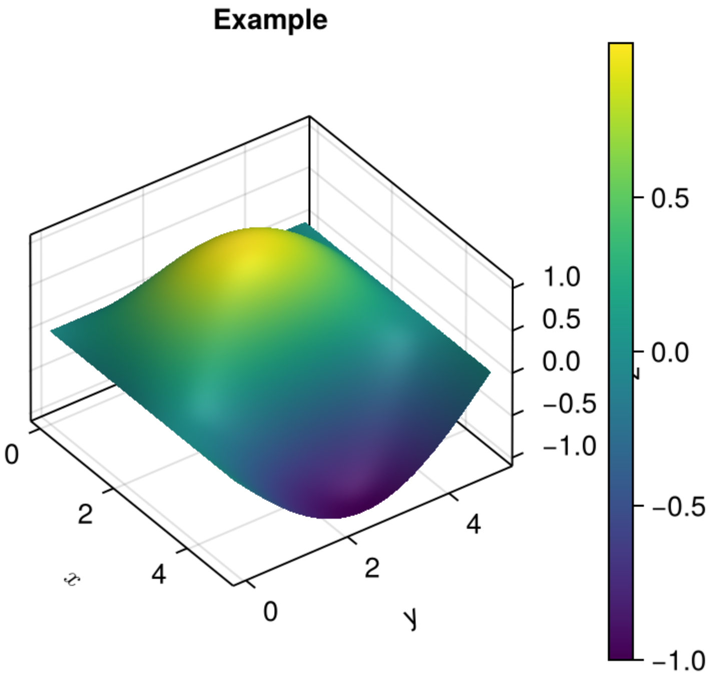
2D plots

k: l:



```
let
    X=0:0.05:5
    Y=X
    fig=Figure(size=(400,400))
    axis=Axis(fig[1,1],xlabel=L"x",ylabel="y",title="Example")
    F=[sin(k*π*X[i])*sin(l*π*Y[j]) for i=1:length(X), j=1:length(Y)]
    cf=contourf!(axis,X,Y,F,levels=64)
    CairoMakie.contour!(axis,X,Y,F,levels=5,color=:black,labels=true)
    Colorbar(fig[1,2],cf)
    fig
end

```

azim: elev:

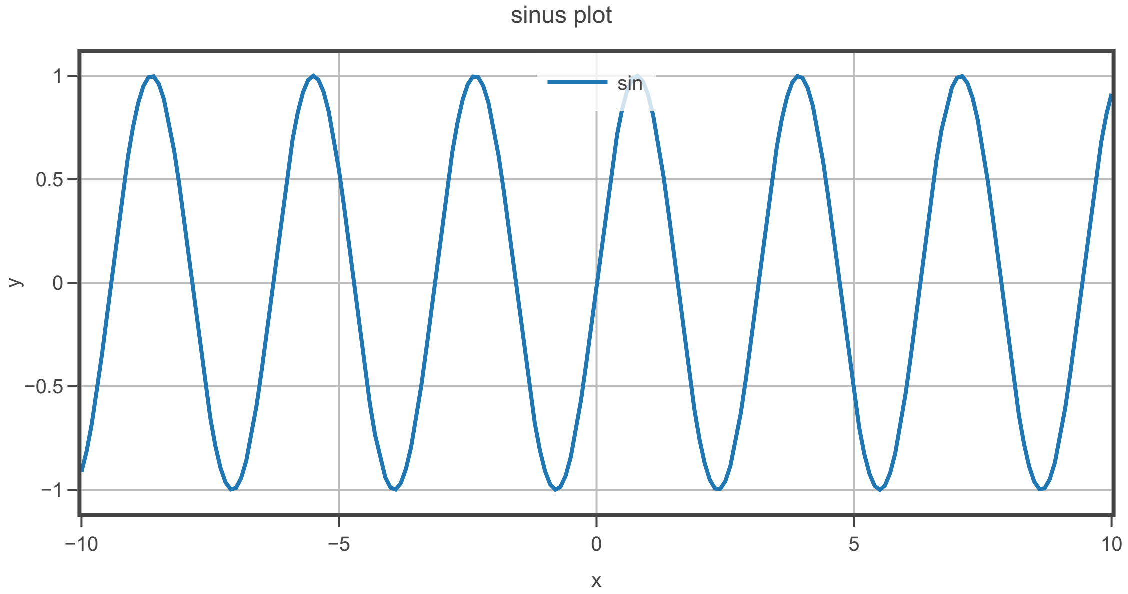


```julia
let
    X=0:0.05:5
    Y=X
    fig=Figure(size=(400,400))
    axis=Axis3(fig[1,1],xlabel=L"x",ylabel="y",title="Example",
        azimuth=azim, elevation=elev)
    F=[sin(k*π*X[i])*sin(l*π*Y[j]) for i=1:length(X), j=1:length(Y)]
    s=surface!(axis,X,Y,F)
   Colorbar(fig[1,2],s)
    fig
end
```

PlutoVista

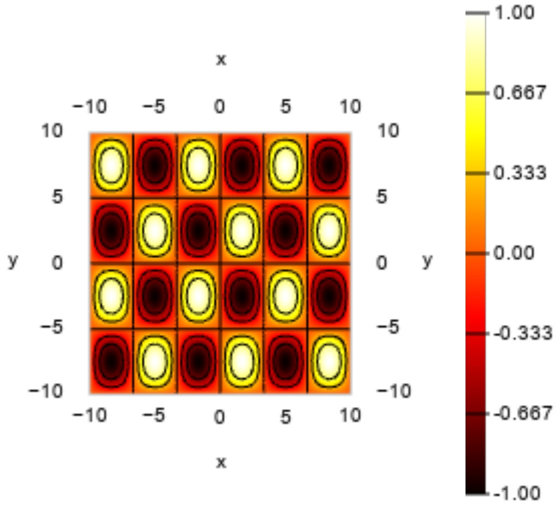
I created PlutoVista.jl for fast plotting in pluto notebooks. For 1D plots, PlutoVista calls back to Plotly.js, and for 2D/3D plots it uses vtk.js, a visualization library for grid and volume data using WebGL as backend.

```julia
using PlutoVista
```



```
PlutoVista.plot(X,sin.(2X),xlabel="x",ylabel="y",label="sin",legend=:ct,
resolution=(600,300), title="sinus plot")
```

```
F2=[sin(k2*π*X[i])*sin(l2*π*X[j]) for i=1:length(X), j=1:length(X)];
```



```
PlutoVista.contour(X,X,F2,size=(400,350),levels=5,colormap=:hot)
```

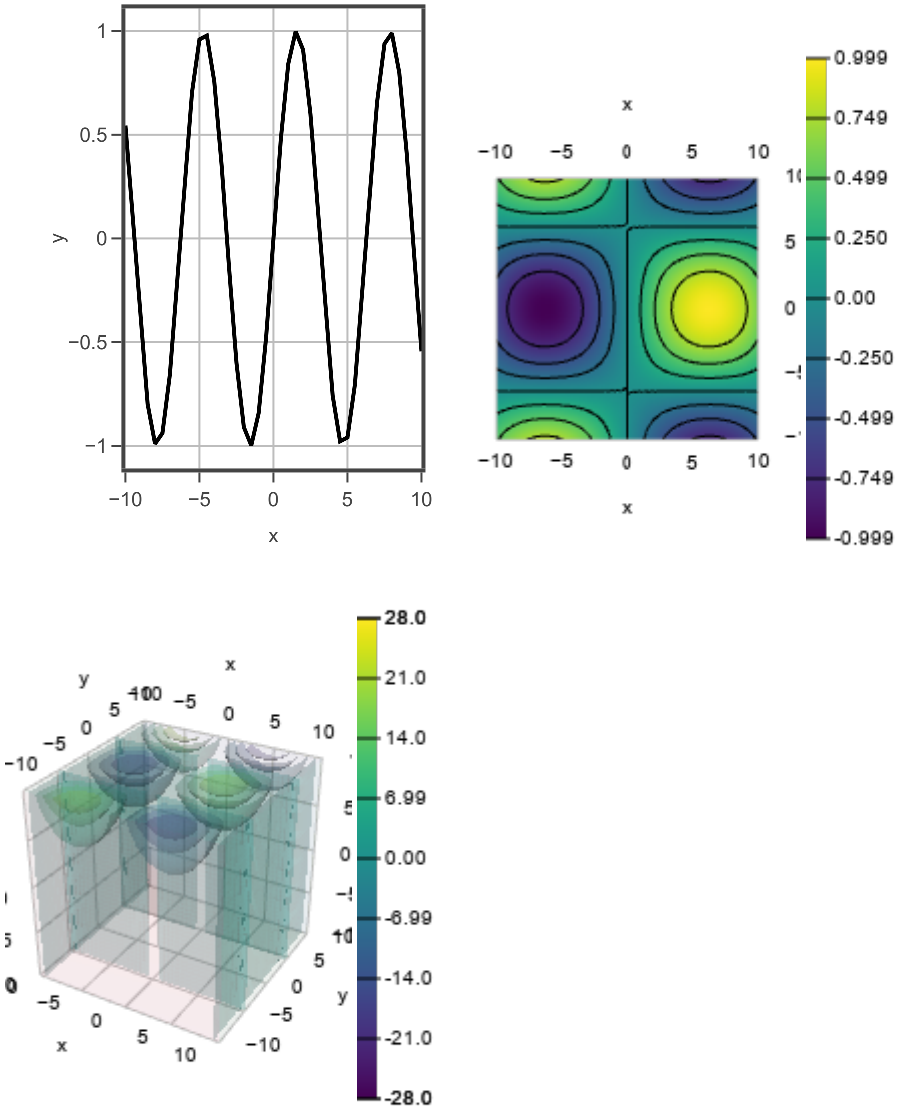
GridVisualize.jl

The GridVisualize.jl package focuses on PlutoVista (for notebooks) and Makie as backends. It is tailored to the visualization of solutions of partial differential equations on 1D/2D/3D grids (of simplices). The idea is to allow the same syntax for different space dimensions.

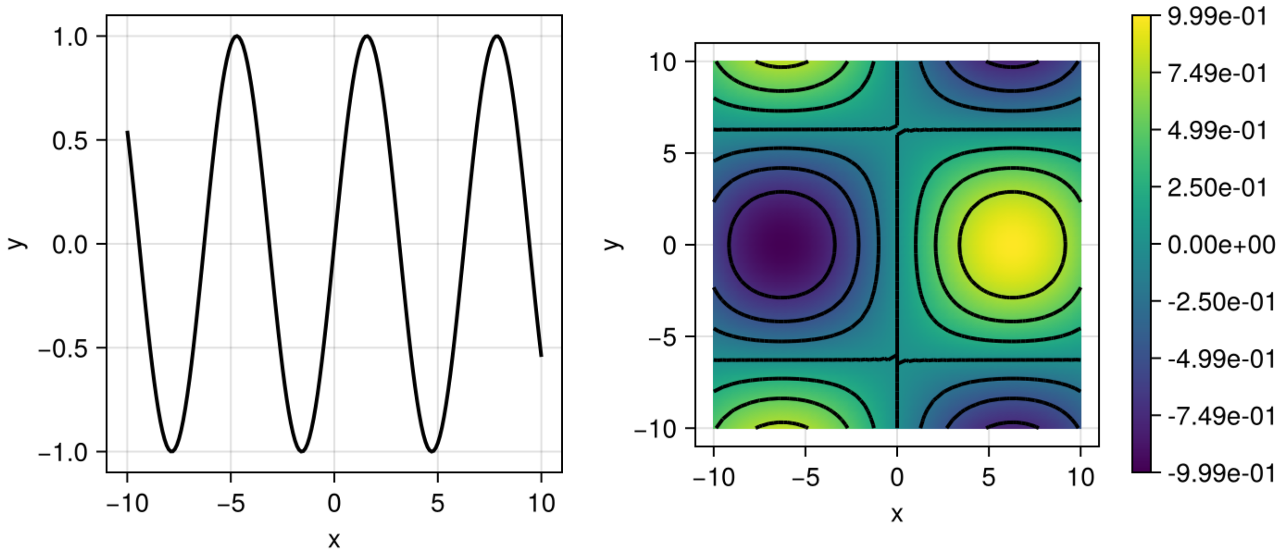
```
using ExtendableGrids, GridVisualize
```

```
X1 = -10.0:0.5:10.0
```

```julia
X1=-10:0.5:10
```



```julia
let
    vis=GridVisualizer(;Plotter=PlutoVista,layout=(1,3),size=(700,300))

    g1=simplexgrid(X1)
    f1=map(sin,g1)
    scalarplot!(vis[1,1],g1,f1)

    g2=simplexgrid(X1,X1)
    f2=map((x,y)->(sin(x/4)*cos(y/4)),g2)
    scalarplot!(vis[1,2],g2,f2)

    g3=simplexgrid(X1,X1,X1)
    f3=map((x,y,z)->(sin(x/3)*cos(y/3)*exp(z/3)),g3)
    scalarplot!(vis[1,3],g3,f3)

    reveal(vis)
end

```



```
let
    vis=GridVisualizer(;Plotter=CairoMakie,layout=(1,3),size=(700,300))

    g1=simplexgrid(X1)
    f1=map(sin,g1)
    scalarplot!(vis[1,1],X,x->sin(x))

    g2=simplexgrid(X1,X1)
    f2=map((x,y)->(sin(x/4)*cos(y/4)),g2)
    scalarplot!(vis[1,2],g2,f2)


    reveal(vis)
end

```

For 3D graphics, the GLMakie backend is better suited.

Plots.jl

Plots.jl: General purpose plotting package with different backends

- GPU support via default `gr` backend (based on "old" OpenGL)
- Support of interactivity in the browser via `plotly` javascript backend
- Precompilation time significantly improved over the last 2 years
- Allows to create publication quality plots
- Partially missing support for triangular grids

GridVisualize.jl partially supports Plots.jl

PyPlot.jl

- PyPlot.jl: Interface to python/matplotlib

  - Realization via PyCall.jl
  - Full functionality of matplotlib
  - Allows to create publication quality plots
  - Also as backend for Plots.jl
  - Problem: slow - most code in python, no support for GPU acceleration
  - Needs to access python installation
- Resources:
  - Julia package
  - Julia examples
  - Matplotlib documentation

GridVisualize.jl partially supports PyPlot.jl

PythonPlot.jl

- PythonPlot.jl: Interface to python/matplotlib
  - Realization via PythonCall.jl

Table of Contents